# Implementing Virtual Peripheral Timers

## Introduction

This application note presents programming techniques for creating multiple, programmable, 16-bit timers/frequency outputs which take advantage of the SX's internal interrupt feature to allow background operation of the timers as virtual peripherals. This example uses the Parallax demo board, taking advantage of Parallax' *SX demo* software user interface and UART features to allow the SX to communicate simply and directly with a personal computer via a serial RS232C port.

## How the circuit and program work

The circuit is extremely simple, requiring one port pin configured as an output for each timer/frequency output. In this implementation, the first timer delivers a square wave frequency output to the LED (port pin RB.6), and uses a current limiting resistor in series with the LED to avoid drawing too much current[1] from the SX and burning out either it or the LED. The other timer controls the speaker (port pin RB.7), which is connected in series with a 10uF capacitor that provides AC coupling and which acts as a high pass filter[2] in conjunction with the speaker's internal resistance. Essentially, this allows the speaker to function normally while blocking direct current flow between the SX and ground that would burn out either the speaker or the SX.

For the LED timer, the interrupt code segment simply adds the timer value to a 16 bit storage accumulator (using two 8 bit registers *timer_accl* + *timer_acch*) on each pass through the interrupt, and then copies the high bit of the result (bit 7 of *timer_acch*) to the LED pin output. This technique allows the 16 bit value stored in (*timer_low* + *timer_high*) to act as a frequency control (rather than a period or counter[3] control) for the square wave output at the LED pin. The code that outputs frequency on the speaker acts identically.

The period of the square wave output can be calculated by multiplying the number of interrupt passes per timer cycle by the period between interrupt passes as follows:

$$period_{int} = mode * prescaler * RETIW\ value^{**} / osc.\ frequency, \quad where\ mode=1\ (turbo)\ or\ =4\ (normal)$$

So, for a crystal frequency of 50 MHz, in turbo mode, with a prescaler of 1, and with an RETIW value of 163, the frequency[4] of the signal at the timer output is:

$$f_{output} = 1 / period_{timer} = (timer\ value / 2^{16}) / period_{int}$$

$$= timer\ value * 50\ MHz / (2^{16} * 1 * 1 * 163) = timer\ value * 0.214\ Hz$$

---

[1]The 470Ω LED resistor limits current to the LED, which at 100% duty cycle draw a current of $I_{LED}$ = Vdd/R = 5V/220Ω = 23mA. The values of this resistor may be adjusted to reduce/increase the overall brightness, though the 30 mA source/sink maximum port current limit should be kept in mind.

[2]The low-frequency cut-off for this filter is far below audio levels.

[3]The code as designed is short and simple, though it lacks the high accuracy of a true 16-bit period counter which can provides accurate timer/frequency output up to the resolution on the oscillator used (which for crystal oscillators is usually very high).

[**]The interrupt is triggered each time the RTCC rolls over (counts past 255 and restarts at 0). By loading the OPTION register with the appropriate value, the RTCC count rate is set to some division of the oscillator frequency (in this case they are equal), which is the external 50 MHz crystal in this case. At the close of the interrupt sequence, a pre-defined value is loaded into the W register using the RETIW instruction which determines the period of the interrupt in RTCC cycles.

[4]Signal frequency is equal to the inverse of the period, i.e. f = 1 / period

The timer's frequency resolution varies depending on the value loaded into the timer, and decreases as the timer value increases. If timing output resolution is a critical factor, all efforts should be made to make sure that any code executed in the interrupt prior to the timer code section maintains a uniform execution rate at all times. This can be done by placing the timer routine before any varying-execution-rate, state-dependent code (it should always come before the UART, for instance), as in the current example.